Simple Window-based Reliable Data Transfer in C/C++

Aaron Cheng
003 933 507

Guan Zhou
604 501 750

Goal and Instructions

The purpose of this project was to use UDP Socket and C/C++ programming language to implement a reliable data transfer protocol. We implemented a Go-Back-N (GBN) protocol. The receiver will first send a message to the sender which includes the name of the file requested. If the file exists, the sender will divide the entire file into multiple packets (the maximum packet size is 1K bytes), and then add some header information to each packet before sending them to the receiver. The header information includes sequence numbers, acknowledgement numbers, and some messages.

Compiling and Running

The Makefile is included in the same folder as packet.h, sender.c and receiver.c. If the source code is modified, one only needs to remove the old executables by running "make clean" and to rebuild the new executables by running "make".

The command to run the sender is as follows:
./sender <port number> <cwnd> <pl> <pc>
The parameters correspond to the port number, congestion window size, probability of loss, and probability of corruption. The probabilities are in the range from 0 to 1.
For example, we can run the server like this:
./sender 9930 5 0.1 0.1

The command to run the receiver is as follows:
./receiver <hostname> <portnumber> <filename> <pl> <pc>
In this project we use localhost, or 127.0.0.1 as hostname. Filename is the name of the file receiver requests.
For example, we can run the receiver like this:
./receiver 127.0.0.1 9930 jpeg.jpeg 0.1 0.1 - or -
./receiver localhost 9930 jpeg.jpeg 0.1 0.1

Receiver & Sender Implementation

The first step to our implementation was to create a header file (packet.h) to define a packet struct that can be used by both sender and receiver. The packet struct contains a packet's type, sequence number, ack number, data size, corrupt state, and a data array to hold the packet data payload.

The receiver takes the hostname, port number of the sender, the name of a file it wants to retrieve from the sender, loss probability, and corruption probability as command line arguments. It opens a socket to construct and send a file request message to the server. If the requested file is found by the server, the receiver begins to receive the data. It looks at the sequence number of the received packets, and if it is correct, it responds with a corresponding ack packet and writes the packet's data to the file. If the received packet is corrupted or if the sequence number is greater than the expected sequence number, the packet is ignored. If the sequence number is less than the

expected number, it responds again with the most recent ack. When a fin packet is received, it responds with a fin ack.

The sender takes the port number, window size, packet loss probability, and packet corruption probability as arguments. It creates a socket and binds it to the chosen port. It then waits for a file request from a receiver. Once a file requested, and the file exists, it begins to transmit the file by dividing it into packets, each holding 1024 bytes of data. It will send as much packets as needed to transfer the file or as much as the congestion window allows. If no acks are received within a given time period, it will resend those packets. If a received packet is corrupted or the ack number is less than what is expected, the packet is ignored. Otherwise, it will slide the window over to the next sequence numbers and send whatever is allowed by the congestion window. This continues until the file is transferred, at which point it will send a fin to the receiver. After receiving a fin ack from the receiver, the connection terminates and it awaits more requests.

Timeout

The implementation of timeout uses the header file “time.h” and FD_SET(). Timeval is a structure that includes at least the following members:

time_t	tv_sec	seconds
suseconds_t	tv_usec	microseconds

Specified time fields tv_sec and tv_usec are used to show how long we wait before timing-out. select() allows our program to monitor multiple file descriptors, waiting until one or more of the file descriptors become "ready" for some class of I/O operation (e.g., input possible). In this project, the select() function enables the sender to realize if the socket has received an ack from the receiver. If no acks are received within the given time value, this constitutes a timeout.

Packet Loss & Corruption

The probabilities of both packet loss and packet corruption are in the range from 0 to 1, and they are handled in a similar way. We use the function rand() to generate a random number, and divide it by RAND_MAX, giving us a fraction ratio. The packet is “lost” and is not sent if the ratio falls within the preset packet loss probability. Similarly, when sending a packet, the packet becomes corrupted if the ratio falls within the preset packet corrupt probability. A value of “1” in the packet’s header indicates that the packet is corrupt and a value of “0” is not corrupt. The sender and receiver both ignore corrupted packets.

Difficulties Faced

Unlike Project 1, getting started with Project 2 took us a while because there was no skeleton code provided. However, upon searching the Internet, we came across http://www.binarytides.com/programming-udp-sockets-c-linux/, which provided a demo of a simple UDP server and client. We played with this demo to figure out how to send

and receive from the same socket using the UDP protocol in both the sender and receiver. This became our skeleton code and our starting point, from which we ultimately used a loop to transfer the requested file.

We also had trouble with how to handle timeouts. Initially, we were not sure how detect timeouts. However, we came across an example when searching the Internet again: https://www.gnu.org/software/libc/manual/html_node/Waiting-for-I_002fO.html. This taught us how to use fd_set and a timeval struct in order to detect if a file descriptor received some sort of input. Using this information, we were able to implement the timeout detection.

Conclusion

This project gave us a good chance to review the GBN knowledge that we learned from class and deepen our understanding of GBN. It was a great experience to apply our previous knowledge of the GBN protocol and to realize it in a C program.